Swinburne University of Technology

Faculty of Science, Engineering and Technology

FINAL EXAM COVER SHEET

Subject Code: COS30008
Subject Title: Data Structures & Patterns
Due date: June 3, 2021, 13:00
Lecturer: Dr. Markus Lumpe

Your name:____________________________ **Your student id:**________________

Check Tutorial	Wed 08:30	Wed 10:30	Wed 16:30	Thurs 08:30	Thurs 10:30	Thurs 14:30	Thurs 16:30	Fri 08:30	Fri 10:30	Fri 14:30

Marker's comments:

Problem	Marks	Time Estimate in minutes	Obtained
1	50	20	
2	54	15	
3	42	10	
4	60	15	
5	8+128=136	60	
Total	342	120	

This test requires approx. 2 hours and accounts for 50% of your overall mark.

Problem 1 (50 marks)

Answer the following questions in one or two sentences:

a. How can we construct a tree where all subtrees have the same degree? (4 marks)

1a)

b. What are reference data members and how do we initialize them? (2 marks)

1b)

c. What is the difference between l-value and r-value references? (6 marks)

1c)

d. What is an object adapter? (6 marks)

1d)

e. What is a key concept of an abstract data types? (4 marks)

1e)

f. How do we define mutual dependent classes in C++? (4 marks)

1f)

g. What must a value-based data type define in C++? (2 marks)

1g)

h. What is the difference between copy constructor and assignment operator and how do we guarantee safe operation? (8 marks)

1h)

i. What is the best-case, average-case, and worse-case for a lookup in a binary tree? (6 marks)

1i)

j. You are given a set of n-1 numbers out of n numbers. How do we find the missing number n_k, $1 \leq k \leq n$, in linear time? (8 marks)

1j)

3-ary Trees and Postfix Traversal

We wish to define a generic 3-ary tree in C++. We shall call this data type `TTree`. A 3-ary tree has a key `fKey` and three nodes `fLeft`, `fMiddle`, and `fRight`. Following the principles underlying the definition of general trees, a 3-ary tree is a finite set of nodes and it is either

- an empty set, or
- a set that consists of a root and exactly 3 distinct 3-ary subtrees.

Somebody has already started with the implementation and created the header file `TTree.h`, but left the project unfinished.

```
#pragma once

#include <stdexcept>

template<typename T>
class TTreePostfixIterator;

template<typename T>
class TTree
{
private:
  T fKey;
  TTree<T>* fLeft;
  TTree<T>* fMiddle;
  TTree<T>* fRight;

  TTree() : fKey(T())      // use default constructor to initialize fKey
  {
    fLeft = &NIL;          // loop-back: The sub-trees of a TTree object with
    fMiddle = &NIL;        //            no children point to NIL.
    fRight = &NIL;
  }

  void addSubTree( TTree<T>** aBranch, const TTree<T>& aTTree )
  {
    if ( !(*aBranch)->empty() )
    {
      delete *aBranch;
    }

    *aBranch = const_cast<TTree<T>*>(&aTTree);
  }

public:
  using Iterator = TTreePostfixIterator<T>;

  static TTree<T> NIL;    // sentinel

  // getters for subtrees
  const TTree<T>& getLeft() const { return *fLeft; }
  const TTree<T>& getMiddle() const { return *fMiddle; }
  const TTree<T>& getRight() const { return *fRight; }

  // add a subtree
  void addLeft( const TTree<T>& aTTree ) { addSubTree( &fLeft, aTTree ); }
  void addMiddle( const TTree<T>& aTTree ) { addSubTree( &fMiddle, aTTree ); }
  void addRight( const TTree<T>& aTTree ) { addSubTree( &fRight, aTTree ); }

  // remove a subtree, may through a domain error
  const TTree<T>& removeLeft() { return removeSubTree( &fLeft ); }
  const TTree<T>& removeMiddle() { return removeSubTree( &fMiddle ); }
  const TTree<T>& removeRight() { return removeSubTree( &fRight ); }
```

```
// Problem 1: TTree Basic Infrastructure

private:

  // remove a subtree, may through a domain error
  const TTree<T>& removeSubTree( TTree<T>** aBranch );

public:

  // TTree l-value constructor
  TTree( const T& aKey );

  // destructor (free sub-trees, must not free empty trees)
  ~TTree();

  // return key value, may throw domain_error if empty
  const T& operator*() const;

  // returns true if this TTree is empty
  bool empty() const;

  // returns true if this TTree is a leaf
  bool leaf() const;

// Problem 2: TTree Copy Semantics

  // copy constructor, must not copy empty TTree
  TTree( const TTree<T>& aOtherTTree );

  // copy assignment operator, must not copy empty TTree
  TTree<T>& operator=( const TTree<T>& aOtherTTree );

  // clone TTree, must not copy empty trees
  TTree<T>* clone() const;

// Problem 3: TTree Move Semantics

  // TTree r-value constructor
  TTree( T&& aKey );

  // move constructor, must not copy empty TTree
  TTree( TTree<T>&& aOtherTTree );

  // move assignment operator, must not copy empty TTree
  TTree<T>& operator=( TTree<T>&& aOtherTTree );

// Problem 4: TTree Postfix Iterator

  // return TTree iterator positioned at start
  Iterator begin() const;

  // return TTree iterator positioned at end
  Iterator end() const;
};

template<typename T>
TTree<T> TTree<T>::NIL;
```

There are actual two template classes here: `TTree<T>` and `TTreePostfixIterator<T>`. The two template classes occur mutually dependent. However, as long as we do not use the iterator elements template class `TTree<T>` can be safely implemented. The C++ compiler ignores unimplemented features that are not used.

The implementation of `TTree<T>` is defined in three stages: basic infrastructure, copy control and, move semantics.

Once these stages are completed, we can focus our attention on the postfix iterator part.

Problem 2 (54 marks)

Implement the basic `TTree<T>` infrastructure:

- `const TTree<T>& removeSubTree( TTree<T>** aBranch );`
- `TTree( const T& aKey );`
- `~TTree();`
- `const T& operator*() const;`
- `bool empty() const;`
- `bool leaf() const;`

Use the available information to implement these features. The method `removeSubTree` has to guarantee that empty trees are not removed. In this case, `removeSubTree` has to throw a domain error. If the subtree can be removed, then a constant reference to it must be returned. In addition, the pointer of the subtree being removed must be set the address of `NIL` to indicate that this branch is now empty.

To create `TTree<T>` objects, we need to define its constructor, and the destructor releases the memory associated with `TTree<T>` objects. The empty tree must not be deleted. It is unique and system-created.

In addition, there are three service functions: `operator*()`, `empty()` , and `leaf()` that return the payload of a `TTree<T>` object, test whether the current `TTree<T>` object is the empty tree, and whether the current `TTree<T>` object is a leaf node, respectively.

You can use `#define P1` in `Main.cpp` to enable the corresponding test driver, if you wish to compile and test your solution. The test driver should produce the following output:

```
Test P1:
The payload of tree: A
The payload of tree.getLeft():   B
The payload of tree.getRight():  C
nD is a leaf node.
Exception: Empty TTree encountered.
Test P1 complete.
```

No other outputs or errors should occur. The method `removeSubTree()` works, if at the end, when object `tree` goes out of scope, no runtime errors occur.

Problem 3 (42 marks)

Implement copy control for `TTree<T>`:

- `TTree( const TTree<T>& aOtherTTree );`
- `TTree<T>& operator=( const TTree<T>& aOtherTTree );`
- `TTree<T>* clone() const;`

Use the available information to implement these features.

The copy control must not create copies of empty trees. If an empty tree is encountered in the copy constructor or assignment operator, then a domain error must be thrown. The method `clone()` can easily prevent copies of empty trees by returning the `this` object. You may need to apply suitable casts where necessary to make the implementation sound.

You can use `#define P2` in `Main.cpp` to enable the corresponding test driver, if you wish to compile and test your solution. The test driver should produce the following output:

```
Test P2:
Copy constructor appears to work properly.
Assignment appears to work properly.
Exception: Copying NIL.
Clone appears to work properly.
Test P2 complete.
```

No other outputs or errors should occur. If the destructor and the elements of copy control work, then, at the end, when objects `tree` and `copy` go out of scope they are properly destroyed.

Problem 4 (60 marks)

Implement move semantics for `TTree<T>`:

- `TTree( T&& aKey );`
- `TTree( TTree<T>&& aOtherTTree );`
- `TTree<T>& operator=( TTree<T>&& aOtherTTree );`

Use the available information to implement these features.

Move semantics avoids copying data when possible. We achieve move semantics by "stealing" the memory associated with the objects being moved. Move semantics uses r-value references. If you use an l-value as an argument to a move operation, then we should find that l-value empty after the move operation. We can use this feature to test out implementation.

You can use `#define P3` in `Main.cpp` to enable the corresponding test driver, if you wish to compile and test your solution. The test driver should produce the following output:

```
Test P3:
std::move makes tree a leaf node.
The payload of tree: A
The payload of tree.getLeft():   B
The payload of tree.getRight():  C
std::move makes copy a leaf node.
The payload of tree: A
The payload of tree.getLeft():   B
The payload of tree.getRight():  C
Exception: Moving NIL.
Test P3 complete.
```

No other outputs or errors should occur. When objects `tree` and `copy` go out of scope they are properly destroyed.

Problem 5 (136 marks)

We now wish to add a postfix iterator to `TTree<T>`. Consider the following figure:

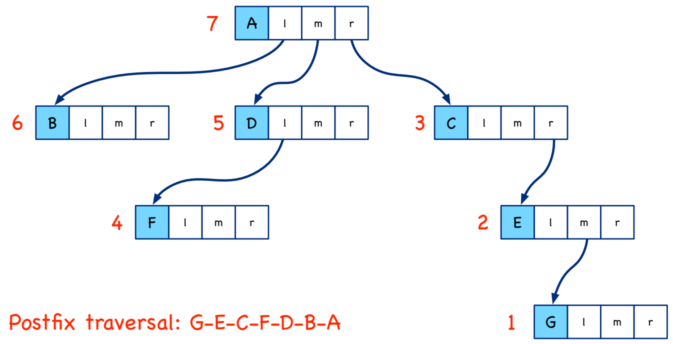

Figure 1: Postfix Traversal.

It depicts a 3-ary tree with 7 nodes, in which the nodes with the payloads "B", "F", and "G" are leafs. Postfix traversal for 3-ary trees first visits the right subtree, then it traverses the middle subtree, followed by the left subtree. Finally, the root is processed. The above figure depicts the traversal order and the sequence in which the payloads would be processed.

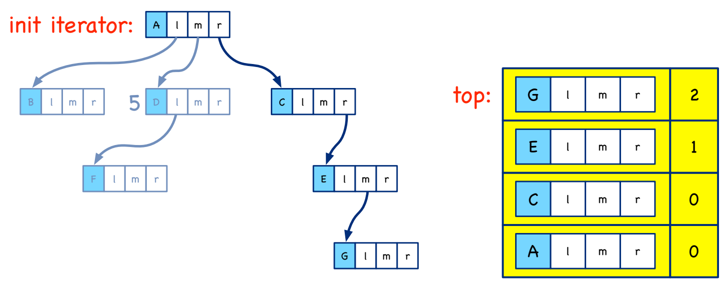

Figure 2: Postfix Traversal Start.

To implement postfix traversal via an iterator, we need two additional data structures: a traversal stack of type `std::stack<Frontier>` and a record structure `Frontier` that stores a pointer to a `TTree<T>` node and an integer that records which subtree of the node we have already visited. Consider Figure 2, which illustrates the traversal stack (on the right) being created by the constructor of the postfix iterator. The bottom slot contains the `Frontier` of the root node "A" of the tree and the integer 0 to indicate that we followed the right subtree. The second last element represents node "C" for which we processed also the right subtree. The second stack entry from the top maps node "E". It does not have a right subtree, but a middle one. The top element stands for node "G", which is a leaf. It's `Frontier` value is 2 – no more subtrees to process. Hence, the number in `Frontier` increases from 0 to 2 when we move from the right subtree to the left subtree via the middle subtree in the corresponding `TTree<T>` node.

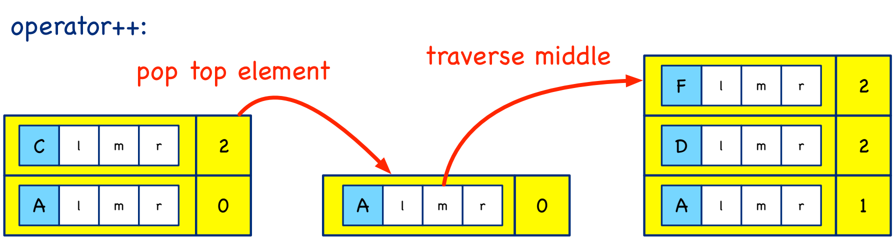

Figure 3: Increment Scenario.

Figure 3 shows what happens when we increment the iterator and node "C" is the top element. Its `Frontier` gets removed from the stack. The `Frontier` for node "A" becomes the top element. We still have not processed all subtrees. There is a middle subtree that needs to be traversed. Hence, we follow the middle subtree and push the `Frontier`s for the nodes "D" and "F". This process completes when all nodes have been visited.

A suitable solution for a postfix iterator is given below:

```
#pragma once

#include "TTree.h"

#include <stack>

template<typename S>
struct TTreeFrontier
{
  size_t stage;                         // frontier stages: 0, 1, 2
  const TTree<S>* node;                 // frontier TTree node

  TTreeFrontier( const TTree<S>* aNode ) :
        node(aNode),                      // TTree node
        stage(0)                          // 0 - start right
    {}
};

template<typename T>
class TTreePostfixIterator
{
private:
  const TTree<T>* fTTree;               // 3-way tree
  std::stack<TTreeFrontier<T>> fStack;  // DFS traversal stack

  using Frontier = TTreeFrontier<T>;

  // push subtree starting with aNode
  void push_nodes( const TTree<T>* aNode );

public:
  using Iterator = TTreePostfixIterator<T>;

  Iterator operator++(int)
  {
    Iterator old = *this;
    ++(*this);
    return old;
  }

  bool operator!=( const Iterator& aOtherIter ) const
  {
    return !(*this == aOtherIter);
  }

  // iterator constructor
  TTreePostfixIterator( const TTree<T>* aTTree );

  // iterator dereference
  const T& operator*() const;

  // prefix increment
  Iterator& operator++();

  // iterator equivalence
  bool operator==( const Iterator& aOtherIter ) const;

  // auxiliaries
  Iterator begin() const;
  Iterator end() const;
};
```

Template class `TTreePostfixIterator<T>` defines a standard forward iterator. To facilitate its implementation there is a private member function `push_nodes()`. This function takes a pointer to a `TTree<T>` object and pushes a corresponding `Frontier` for it and its rightmost subtrees, if there are any. See Figure 2 for a guide on the process.

The prefix increment always removes the top element from the stack. This is a feature of postfix traversal. Next, if the traversal stack is not empty, we have to inspect the `Frontier` of top. We have to check if there are still subtrees to process and push the next subtree onto the stack, if such a subtree exists.

The other iterator methods are defined in the usual way. The constructor has to set up the initial stack. That is, for our sample scenario, the `Frontier` for node "G" must be the top element, once the constructor has completed.

Please note template class `Frontier<S>` is defined as a `struct`. That is, it defines a class where all members by default have public access.

To complete the solution, you need to implement the iterator methods for class `TTree<T>`. They are used to map for-range loops to plain for loops in C++. The compiler will report "undefined symbol" if these methods have not been implemented.

You can use `#define P4` in `Main.cpp` to enable the corresponding test driver, if you wish to compile and test your solution. The test driver should produce the following output:

```
Test postfix iterator: G E C F D B A
```

No other outputs or errors should occur.